# An Approach for Formal Verification of Updated Java Bytecode Programs

Razika Lounas[1,2]
[1]University of M'hamed Bougara of Boumerdes
Facutly of Sciences, LIMOSE Laboratory
Avenue de l'independance, 35000 Boumerdes
Algeria
[2]University of Limoges
123 Avenue Albert Thomas, 87700 Limoges, France
*razika_lounas@umbb.dz*

Mohamed Mezghiche
University of M'hamed Bougara of Boumerdes
Facutly of Sciences, LIMOSE Laboratory
Avenue de l'independance, 35000 Boumerdes
Algeria
*mohamed-mezghiche@umbb.dz*

Jean-Louis Lanet
INRIA LHS-PEC
263 Avenue Général Leclerc, 35000 Rennes
France
*jean-louis.lanet@inria.fr*

**This paper deals with formal specification and verification of Java bytecode update. Programs update for java applications has gained a wide interest since it is used for several purposes: transforming semantics of a program, adding features to a program or performing optimizations. In this paper, we focus on program transformations for java programs at the bytecode level. Because these transformations may introduce errors, our goal is to provide a formal way to verify the update and establish its correctness. Our approach for formal specification and verification of updated Java bytecode programs is based on four ingredients: a formal interpretation of the semantics of update operations, a functional representation of bytecode, bytecode annotation and predicate transformation calculus. We use the concept of Hoare predicate transformation to derive a specification of an annotated bytecode. Annotations are used to express update operations within the code. A functional representation is used to model annotations and bytecode. The approach derives then a new specification for the annotated bytecode using a weakest precondition calculus defined to deal with update operations. Verification conditions are then generated and proved to establish the correction of the update.**

*Bytecode transformation, formal semantics, weakest precondition calculus, bytecode verification.*

## 1. INTRODUCTION

During their life cycle, programs need to be updated in order to alter their semantics, perform optimizations or add features. Several techniques were presented for this purpose in literature, for example, (Neamtiu et al. (2006) and Gupta et al. (1996)) present systems for C programs updating and (Orso et al. (2002), Hlopko et al. (2013)) present systems to update Java programs.

Updating programs leads to the transformation of their elements such as code, data structures and state. We focus on the transformation of Java codes. In this context, several tools were developed, for example, Java Syntactic Extender (JSE) (Bachrach and Playford (2001)) and ixj (Boshernitsan and Graham (2004)). However, in some cases, the source code is not available (or not distributed). Transforming a program at bytecode level is an interesting alternative since several languages like Java or Java Card are based on virtual machines executing bytecode. Transforming programs at bytecode level offers some advantages: it does not require to recompile which can be a time consuming task as in the case of transformations at source code level. On the other hand, bytecode level transformation is more complex than source-level manipulation for the users because they have to know bytecode language very well and because of the many low-level details one needs to use.

Java bytecode transformation is used in several applications and several tools were developed to manipulate Java bytecode programs such as BCEL

(Dahm (1999)) and RuggedJ (McGachey et al. (2009)). In (Sakamoto et al. (2000)), the authors developed an algorithm to ensure portable thread migration in Java. This algorithm is based on bytecode transformation. Bytecode is transformed in order to enable programs to save and restore their execution state after migration through the network. Another purpose for bytecode transformation is presented in (Binder and Hulaas (2005)) where a framework based on bytecode transformation is developed in order to enable Java applications to perform CPU management.

In some cases, the transformation occurs at runtime. The update is then said to be dynamic (Dynamic Software Update: DSU). In (Noubissi (2011)) and in (Noubissi et al. (2011)), the authors presented a system to perform DSU: while the Java Card virtual machine is executing the program, the bytecode is updated.

This large interest of Java bytecode transformation and its use in many critical applications raise the question of its correctness. In fact, a transformation may introduce an error which may alter the bytecode leading the system to an unexpected state. Besides, in some cases, the update is critical (e.g. EmbedDSU) in such a way that an attacker can take advantage of an incorrect update. In these applications where security issues are involved the update must pass some certification procedure for example Common Criteria (Common Criteria (2015)). For a certain certification level one has to provide a formal proof of the security mechanism implemented. A formal way to specify transformations and verify their correctness is then necessary.

Formal methods offer rigourous means in specifying software properties and establishing the correctness of programs regarding their formal specifications. In this work, we present an approach for formal verification of bytecode update. We focus on Java bytecode and the system presented in (Noubissi et al. (2011)) called embedDSU: a system developed to implement DSU functionalities in Java Card applications. It is based on two parts: off-card in which a module called DIFF generator computes the syntactic changes between the old and the new version of the application and generates a DIFF file (called also a patch). This patch is then sent on the card to perform the update by other modules implemented by extending the Java Card virtual machine.

In this work, we propose to formally verify that the obtained bytecode is semantically equivalent to the one written by the programmer and used to perform the DIFF file. Our approach is based on the following contributions: the definition of a new weakest precondition calculus as the base of the verification process, a formal interpretation of the semantics of the update operations, a functional representation of bytecode programs and bytecode annotation. The choice of functional representation is motivated by our interest in capturing the behavior of the initial bytecode and the updated version and the mature existing tools for formal reasoning about functional programming languages.

This paper is organized as follows: in section 2 we give an overview of embedDSU. Section 3 introduces the language and the formal semantics of the updates. In section 4, we present an overview of our approach in its steps. We present the specification languages is section 5. In section 6, we give our functional modelisation of Java bytecode and annotations. We propose a predicate calculus for update operations in section 7 and give the notion of a correct update. This section ends with an example to show how the logic works. We discuss related work in section 8 and conclude in section 9.

## 2. OVERVIEW OF EMBEDDSU

EmbedDSU (Noubissi (2011), Noubissi et al. (2011), Noubissi et al. (2010)), is a software-based DSU technique for Java-based smart cards which relies on the Java virtual machine. It is based on the modification of an embedded virtual machine. EmbedDSU is divided in two parts: off-card and on-card:

**(i)** In off-card, a module called *DIFF generator* determines the syntactic changes between versions of classes in order to apply the update only to the parts of the application that are really affected by the update. The changes are expressed using a Domain Specific Language (DSL). Then, the DIFF file result is transfered to the card and used to perform the update.

**(ii)** The on-card part is divided into two layers: 1) Application Layer: The binary DIFF file is uploaded into the card. After a signature check with the *wrapper*, the binary DIFF is interpreted and the resulting instructions are transferred to the *patcher* in order to perform the update. The *patcher* initializes data structures for update. These data structures are read by the *updater* module to determine what to update and how to update, by the *safeUpdatePoint detector* module to determine when to apply the update and by the *rollbacker* to determine how to return to the previous version in case of update failure. These points require the introspection of the virtual machine. 2) System Layer: the modified virtual

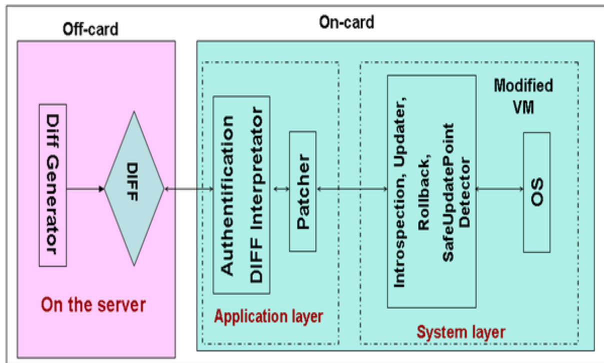


***Figure 1:*** *Architecture of EmbedDSU*

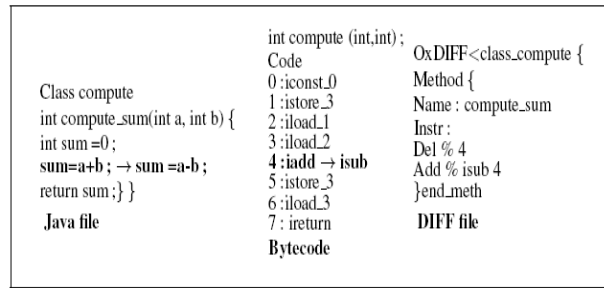


***Figure 2:*** *An example of a patch (DIFF file)*

machine supports the followings features: (1) *Introspection* module which provides search functions to go through VM data structures like the references tables, the threads table, the class table, the static object table, the heap and stack frames for retrieving information necessary to other modules; (2) *updater* module which modifies object instances, method bodies, class metadata, references, affected registers in the stack thread and affected VM data structures; (3) *SafeUpdatePoint detector* module permits to detect safe point in which we can apply the update by preserving coherence of the system.

The system EmbedDSU is suitable for smard cards especially in term of resource limitations. It was established that sending a DIFF file is less ressource consuming than sendig the whole new version to the card and perform updates and that the resources implied by the update modules are acceptable in term of memory occupation (Noubissi (2011)). The system EmbedDSU updates three principal parts:

**(i)** The bytecode: the process updates first the bytecode of the updated class and the meta data associated with it e.g., constant pool, fields table, methods table...

**(ii)** The heap: The process updates the instances of the updated class in the heap, obtains new references for modified objects and updates instances using these references.

**(iii)** The frames: The process updates in each frame in the thread stack the references of updated objects to point to new instances.

This paper addresses the first part: bytecode update at the method level. The types of updates that may occur are: adding, modifying or suppressing bytecode instructions, changing the signatures of a method or modifying local variables. These updates are contained in the DIFF file which indicates the update and where it occurs in the bytecode. An example is shown figure 2: the patch indicates that the instruction *iadd* in the method *compute_sum* is deleted and the instruction *isub* is added at the same place provided by the program counter.

## 3. LANGUAGE AND SEMANTICS

### 3.1. The language

For the definition of the semantics, we extend the formalism used by Freund and Mitchell (Freund and Mitchell (1999)). The authors define a type system for a small subset of Java bytecode. We define a subset and propose to extend it with instructions to indicate updates called update instructions (*Upd_instr*) for instruction addition, deletion and modification. In this definition, $x$ is a local variable; $L$ is an instruction address; $A$ is a class name; $f$ is a field name; $l$ is a method name and $pc$ the program counter.

$Instruction ::= |pop\ |if\ L\ |store\ x\ |load\ x\ |new\ A$
$|binop\ |neg\ |const\ a\ |invokevirtual\ A\ l\ t\ |goto\ L$
$|getfield\ A\ f\ t\ |putfield\ A\ f\ t\ |return$

$Upd_Instr ::= Add_Inst\ Instruction\ pc$
$|Dlt_Inst\ Instruction\ pc$
$|Mod_Inst\ Instruction\ instruction\ pc$

In this language, the instruction $pop$ extracts the top of the stack and $const\ a$ pushes a constant $a$ on the top of the stack. The instruction $load\ x$ pushes the value in the variable $x$ on the top of the stack whereas the instruction $store\ x$ pops the top of the stack and stores it in the variable $x$. The instruction $if\ L$ jumps to $L$ if the top of the stack is not zero else it performs the following instruction. $Goto\ L$ jumps to $L$. The instruction $New\ A$ allocates a new object of type $A$ and pushes it on the top of the stack. The instructions manipulating fields are : $getfield\ A\ f\ t$ and $putfield\ A\ f\ t$. Getfield reads the field $f$, which has the type $t$ of the object of class $A$ whose reference is on the top of the stack and pushes its value on the top of the stack and putfield modifies the field $f$ with the value popped form the stack.

The instruction $invokevirtual$ invokes the method $l$ of signature $t$ and the class $A$. The instruction $Binop$ is used to gather arithmetic binary operations: $add$, $mult$ and $sub$. The instruction $neg$ negates the top of the stack and $return$ is for method return.

Update instructions are respectively: adding an instruction, deleting instruction and modifying an instruction. We indicate the place of the update operation with $pc$.

### 3.2. Operational semantics for bytecode instructions

We model the interpretation of the instructions of the bytecode instructions using the standard framework for operational semantics (Freund and Mitchell (1999), Bannwart and Müller (2005)). Each instruction is characterised by the transformation of a configuration. A configuration $< M, s, h, f, pc >$ representing a step execution consists of an operand stack $s$, a heap $h$, a local variables map $f$, a program counter $pc$ and the body $M$. Operational semantics is defined by a transition relation over configurations. A transition $< M, s, h, f, pc > \rightarrow < M, s2, h2, f2, pc2 >$ takes the state from the configuration $< M, s, h, f, pc >$ to the configuration $< M, s2, h2, f2, pc2 >$.

The rules for the instructions of our language are represented in table 1. The instruction $new\ A$ creates a new object of class A, thereby modifying the current heap. A reference to the new object is pushed onto the stack. $store\ x$ pops a value from the evaluation stack and assigns it to a variable,$f$ is modified accordingly. $load\ x$ put the value of x on the top of the stack. The $binop$ operation which pops two values from the stack, performs the binary operation, and pushes the result. $if\ l$ has two rules; wether it jumps to the indicated line or performs the following instruction according to the value of the top of stack. The instruction $putfield$ updates the heap with the new value of the field of the object which is on the top of the stack. The new value is popped from the second element of the stack. $invokevirtual$ invokes the method $l$ on an object reference and parameters on the stack and replaces these values by the return value $v$ of the invoked method after its execution.

### 3.3. Formal semantics for update instructions

We propose a static semantics to express the effects of update instructions on a configuration of the bytecode. This semantics was introduced in our initial paper (Lounas et al. (2012)). The purpose of the semantics is to express formally the effects and the conditions of update instructions and thus prevent type errors in the updated bytecode. In this paper, we give more rules and show how to use the semantics to establish that an updated program is well typed. It is also used in further section to derive specifications for program transformations. In the rules shown in tables 2 and 3, $F$ is a mapping from a program point to a mapping from a frame variable to a type. $S$ is a mapping from a program point to an ordered sequence of types, $i$ denotes a program point or an address of code. The map $F_i$ gives a type of local variables at program point $i$. The string $S_i$ gives the types of entries in the operand stack at program point $i$. These $F$ and $S$ are useful to our semantics since they contain typing information about valid local variables and entries in the operand stack respectively. $SD$ represents the stack depth and $M$ (mapping) is a function that associates a number to each line. $Dom$ is the set of addresses used by the method. A configuration at line i is represented by $< (F, S, SD, M), i >$. The judgement that expresses that a bytecode $BC$ is well typed by $F$, $S$, $SD$ and $M$ is:

$$\frac{\begin{array}{l} F_1 = F_\top, SD1 = 0 \\ S_1 = \varepsilon, M1 = Map(BC) \\ \forall i \in DOM(BC), F, S, SD, M, i \vdash BC \end{array}}{F,S,M,SD \vdash BC}$$

The first two lines of the judgement represent the initial configuration: all variables are mapped to the value $top$ (default initial value), stack depth is zero, the sequence of types is initially empty ($\varepsilon$) and $M1$ is the mapping of the initial bytecode. The last line expresses that each instruction (update instruction) in the bytecode is well typed. This is ensured by the rules given in tables 2 and 3. For illustration, the insertion of the instruction $new\ A$ at line $i + 1$ allows us to obtain a new configuration if the stack depth is incremented, local variables are not affected and in the stack, the type $A$ is inserted. In the instruction $invokevirtual$ the function $dom$ represents the domain of the invoked function (types of its arguments) and the function $card$ represents the number of elements in the domain. The rule expresses that these arguments are popped from the stack of type and then the result is pushed. For the insertion of an instruction representing an arithmetic binary operation $Binop$, we show the rule of the instruction $add$: this operation pops two elements (integers) from the stack and then pushes the result. $mult$ and $sub$ have analogous explanations by writing the right operation. In the rules, the mapping $M2$ is the result of operations on $M1$. The operations which represent manipulations on bytecode are: *range* and *shift*. The operation *range* extracts from a mapping $M1$ a part $M2$ included between line $n$ and line $m$. The second operation shifts a part from a mapping between $n$ and $m$ for $p$ positions which is determined by the number of added instructions.

We define the operations $look_for_jumps$ and $update_jumps$ to take into account jumps in bytecode transformation: $look_for_jumps$ returns from a mapping a list of jumps instructions represented by their line number and the operation update_jumps updates jump instructions:
$Look_for_jumps : mapping \rightarrow int\ list$
$Update_jumps : mapping * int\ list * int \rightarrow mapping$

These operations updates jumps within the bytecode if necessary. When we add for instance an instruction at $pc$, the instructions after this position are shifted and their numbers change. It is then necessary to update goto and if instructions accordingly. These modifications keep the structure of the bytecode coherent. In the rules for instructions suppression (table 3), $Effect_STK$, $Effect_F$ and $Effects_SD$ are used to express the effects of an instruction of the stack and the local variables and stack depth. They are used to readjust these elements to the instruction at $(i + 1)$ in the new bytecode after the suppression. The notation $(M2)F$ (Respectively, $(M2)S$) is used to express $F$ (Respectively, $S$) in the mapping $M2$. We notice that in this formalisation, a modification is considered as a suppression followed by an insertion.

## 4. APPROACH FOR FORMAL VERIFICATION

The mechanism of EmbedDSU implies the modification of the bytecode of a running application on-card after the conventional verification during the process of its life cycle. In this process, bytecode passes verification process based especially on type verification. The applications of update operations on-card is performed with insertion and suppression of instructions according to the DIFF file. Consequently, we obtain on-card, after the update process, a new bytecode that was not submitted to the conventional verification process. Our framework allows to:

**(i)** Ensure the validity of update operations of the DIFF file according to the formal specification of the Java Card virtual machine specification.

**(ii)** Guarantee that the application of the update leads to a bytecode with the specification that is conform to the intended specification (provided by the programmer).

The first point is ensured by the formalisation of the semantics of update operations. In the second point, we aim to establish that given an initial program $P1$, its new version $P2$ and a DIFF file $\Delta$ containing the specification of the transformation derived from the differences between $P1$ and $P2$, the application of the DIFF file on-card on $P1$ (noted App_PATCH) leads to $P2'$. The two programs $P2$ and $P2'$ are verified to be

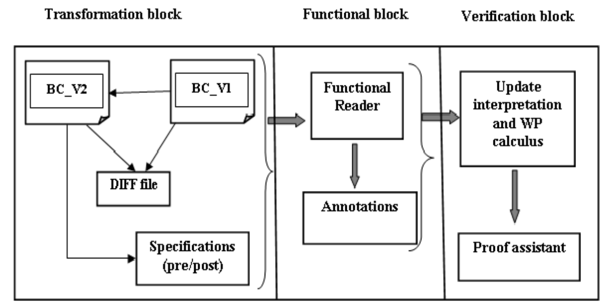


***Figure 3:*** *Approach for verification*

semantically equivalent. This equivalence ensures that the system indeed implemented the desired transformation. This problem can be expressed equationally by:

$$\forall P1, P2, P2', \Delta = DIFF(P1, P2), P2' = App_PATCH(P1, \Delta) \Rightarrow P2 \equiv P2'$$

This raises two major issues: 1- how to model the application of the DIFF file on an existing program? and 2- how to express the equivalence which guarantees the correctness of the update? We present the overview of our approach for transformation verification. Figure 3 represents an overview of our approach which is split in three parts:

**(i)** *The transformation block:* in this stage, we obtain from a first version of a bytecode program $BC_V1$ and a second version $BC_V2$ (Version one transformed), a DIFF file. This DIFF file will be applied to the on-card first version. We obtain a new version on-card. The goal of our approach is to establish that the on-card new version and $BC_V2$ are semantically equivalent. At this level, the specifications of both $BC_V1$ and $BC_V2$ are provided by the programmer using existing specification languages.

**(ii)** *The functional block:* we define a functional model for representing and manipulating the Java Card bytecode. We implement an automatic translator called *functional_reader* which takes a program written in bytecode and produces a functional representation of it. The application of the DIFF file is represented at this level as annotations of the functional representation with expressions indicating the place of the update operation and its nature (addition of instructions, deletion . . . )

**(iii)** *The verification block:* our goal is to verify that the bytecode obtained by transformation is equivalent to the one written by the programmer *i.e.*, it satisfies the same specification. The

***Table 1:*** *Rules for operational semantics*

$$\frac{M[pc]=pop}{<M,v.s,h,f,pc>\rightarrow <M,s,h,f,pc+1>} \qquad \frac{M[pc]=new\ A,h'=h[create(A,ref)]}{<M,s,h,f,pc>\rightarrow <M,ref.s,h',f,pc+1>} \qquad \frac{M[pc]=load\ x}{<M,s,h,f,pc>\rightarrow <M,f[x].s,h,f,pc+1>}$$

$$\frac{M[pc]=store\ x}{<M,v.s,h,f,pc>\rightarrow <M,s,h,f[x\leftarrow v],pc+1>} \qquad \frac{M[pc]=if\ l}{<M,0.s,h,f,pc>\rightarrow <M,s,h,f,pc+1>} \qquad \frac{M[pc]=if\ l,v\neq 0}{<M,v.s,h,f,pc>\rightarrow <M,s,h,f,l>}$$

$$\frac{M[pc]=const\ a}{<M,s,h,f,pc>\rightarrow <M,a.s,h,f,pc+1>} \qquad \frac{M[pc]=getfield\ \ a\ f\ t,v=h[o.f]}{<M,o.s,h,f1,pc>\rightarrow <M,v.s,h,f1,pc+1>} \qquad \frac{M[pc]=neg}{<M,v.s,h,f,pc>\rightarrow <M,(-v).s,h,f,pc+1>}$$

$$\frac{M[pc]=binop,op\in\{+,-,*\}}{<M,v1.v2.s,h,f,pc>\rightarrow <M,(v1\ op\ v2).s,h,f,pc+1>} \qquad \frac{M[pc]=putfield\ A\ f\ t,h'=h[o.f\leftarrow v]}{<M,o.v.s,h,f1,pc>\rightarrow <M,s,h',f1,pc+1>} \qquad \frac{M[pc]=goto\ l}{<M,s,h,f,pc>\rightarrow <M,s,h,f,l>}$$

$$\frac{M[pc]=invokevirtual\ A\ l\ t\ ,<l,\varepsilon,h,f_l,0>\rightarrow<l,v,h1,f_l',pc_l>}{<M,a_1...a_n.s,h,f,pc>\rightarrow <M,v.s,h1,f1,pc+1>}$$

***Table 2:*** *Rules for update operations (insertion of instructions)*

$$\frac{\begin{array}{l} Add_inst\ \ goto\ L(i+1) \\ SD_{i+1}=SD_i\ PC_MAX++ \\ S_{i+1}=S_i\ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,goto\ L,i+1) \\ i+1,L\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC} \qquad \frac{\begin{array}{l} Add_inst\ \ store\ x(i+1) \\ SD_{i+1}=SD_i-1\ \ PC_MAX++ \\ S_i=t.S_0\ \ F_{i+1}=F_i[x\leftarrow t] \\ S_{i+1}=S_0 \\ M2=Add_inst(M1,store\ x,i+1) \\ i+1\in DOM(BC)\ x\in VAR(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC} \qquad \frac{\begin{array}{l} Add_inst\ add(i+1) \\ SD_{i+1}=SD_i-1 \\ S_i=int.int.S_0\Rightarrow \\ S_{i+1}=int.S_0 \\ M2=Add_inst(M1,add,i+1) \\ i+1\in DOM(BC)\ \ F_{i+1}=F_i \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ pop\ (i+1) \\ SD_{i+1}=SD_i-1\ \ F_{i+1}=F_i \\ S_i=t.S_0\rightarrow S_{i+1}=S_0 \\ M2=Add_inst(M1,pop,i+1) \\ PC_MAX++ \\ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC} \qquad \frac{\begin{array}{l} Add_inst\ putfield(A,f,t)(i+1) \\ SD_{i+1}=SD_i-2\ F_{i+1}=F_i \\ S_i=t.A.S_0\Rightarrow S_{i+1}=S_0 \\ M2= \\ Add_inst(M1,putfield(A,f,t),i+1) \\ PC_MAX+3\ \ \ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC} \qquad \frac{\begin{array}{l} Add_inst\ \ new\ A(i+1) \\ SD_{i+1}=SD_i+1 \\ S_{i+1}=A.S_i\ \ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,new\ A,i+1) \\ PC_MAX++ \\ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ getfield(A,f,t)(i+1) \\ SD_{i+1}=SD_i \\ S_i=A.S_0\Rightarrow S_{i+1}=t.S_0 \\ M2= \\ Add_inst(M1,getfield(A,f,t),i+1) \\ PC_MAX+3\ \ \ \ F_{i+1}=F_i \end{array}}{F,S,M2,SD,i+1\vdash BC} \qquad \frac{\begin{array}{l} Add_inst\ \ load\ x(i+1) \\ SD_{i+1}=SD_i+1 \\ PC_MAX++ \\ S_{i+1}=F_i[x].S_i\ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,load\ x,i+1) \\ i+1\in DOM(BC)\ x\in VAR(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC} \qquad \frac{\begin{array}{l} Add_inst\ \ if\ \ L(i+1) \\ SD_{i+1}=SD_i \\ PC_MAX++ \\ S_{i+1}=S_i\ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,if\ L,i+1) \\ i+1,L\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

$$\frac{\begin{array}{l} Add_inst\ \ invokevirtuel(A,l,t)(i+1) \\ SD_{i+1}=SD_i-(card(dom(t))+1) \\ S_{i+1}=tn_1.tn_2\ldots tn_n.S_0\rightarrow S_{i+1}=S_0 \\ M2= \\ Add_inst(M1,invokevirtuel(A,l,t),i+1) \\ i+1\in DOM(BC)\ \ F_{i+1}=F_i \\ PC_MAX+3 \end{array}}{F,S,M2,SD,i+1\vdash BC} \qquad \frac{\begin{array}{l} Add_inst\ \ const\ a(i+1) \\ SD_{i+1}=SD_i+1 \\ PC_MAX++ \\ S_{i+1}=int.S_i\ \ F_{i+1}=F_i \\ M2= \\ Add_inst(M1,const\ a,i+1) \\ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC} \qquad \frac{\begin{array}{l} Add_inst\ \ neg\ (i+1) \\ SD_{i+1}=SD_i\ F_{i+1}=F_i \\ S_i=int.S_0=S_{i+1} \\ M2= \\ Add_inst(M1,negi+1) \\ PC_MAX++ \\ i+1\in DOM(BC) \end{array}}{F,S,M2,SD,i+1\vdash BC}$$

***Table 3:*** *Rules for update operations (suppression of instructions)*

$$\frac{\begin{array}{l} Dlt_inst\ \ goto\ L\ \ (i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, goto\ L, i+1) \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1, L \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (add\ \ (i+1)) \\ M2 = Dlt_inst(M1, add, i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ S_i = int.int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (putfield(A,f,t)\ \ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, putifield(A,f,t), i+1) \\ S_i = A.t.S_0 \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ new\ A\ \ (i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, new\ A, i+1) \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (neg\ \ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, neg, i+1) \\ S_i = int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (const\ a\ \ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, const\ a, i+1) \\ S_i = S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (store\ x\ \ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, store\ x, i+1) \\ S_i = t.S_0, F_i[x] = t \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], t.S_0) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (pop\ \ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, pop, i+1) \\ S_i = t.S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], t.S_0) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (getfield(A,f,t)\ \ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, getifield(A,f,t), i+1) \\ S_i = A.S_0 \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], A.S_0) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ if\ L\ \ (i+1) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, if\ L, i+1) \\ S_i = int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1, L \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (load\ x\ \ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, load\ x, i+1) \\ (M1)S_{i+1} = t.S_0 \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], S_0) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

$$\frac{\begin{array}{l} Dlt_inst\ \ (invokevirtuel(A,l,t)\ \ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects_SD(a, M2[i+1]) \\ M2 = Dlt_inst(M1, invokevirtuel(A,l,t), i+1) \\ S_i = tn_1.tn_2 \ldots tn_n.S_0 \rightarrow \\ (M2)S_{i+1} Effects_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC_MAX -- \end{array}}{F,S,M2,SD,i+1 \vdash BC}$$

specification of the obtained bytecode in its functional representation with annotations is performed by a weakest precondition calculus that we define specially to deal with update operations. A verification condition generator gives then statements to be verified to establish that the obtained specification matches the specification given by the programmer at the level one. A proof assistant is used to discharge verification conditions.

## 5. JML AND BML SPECIFICATIONS

The starting point is a new version $BC_V2$ of un existing program $BC_V1$. First the programmer writes the new version with its specification in terms of pre/post conditions. The specification language used is JML (Java Modeling Language).

JML (Burdy et al. (2005)) is a specification language for Java/Java Card programs. It allows assertions to be included in the source code, specifying for example pre- and postconditions and invariants. JML annotations are a special kind of Java comments: they are preceded by / / @, or written between /* @ and @* /.

A simple method specifications is of the form:

```
/*@ normal_behavior
requires : <precondition> ;
ensures : <postcondition> ;
@*/
```

This specification means that if the precondition (*requires*) holds at the beginning of a method invocation, then the method terminates normally and the postcondition (*ensures*) will hold at the end of the method. Constructs are defined to write assertion such as: $\backslash old$, to denotes the old value of a variable,$\backslash result$ to denote the result of a method and the quantifiers, $\backslash forall$ and $\backslash exists$.

The DIFF file in the system EmbedDSU is created from the program's bytecode. To ensure the correctness of the transformation, the verification of the specification will be done at bytecode level. The language BML (Burdy et al. (2007)), allows to express specifications of bytecode programs. Its formalism is based on JML and the structures of specifications in both languages are very similar.

At the transformation block, specifications for both first version and second version are written in JML. Starting from a specified source code $\{P_{jml}\}code_{source}\{Q_{jml}\}$, with $P_{jml}$ and $Q_{jml}$ representing respectively precondition and postcondition of $code_{source}$, we obtain a specified bytecode program $\{P_{bml}\}code_{BC}\{Q_{bml}\}$. This information is obtained by applying a compiler JML2BML and will be used by the next stages of the approach to perform verification condition generation and ensure the transformation correctness.

```
int compute (int,int) ;
Code
0 :iconst_0
1 :istore_3
2 :iload_1
3 :iload_2
4 :iadd
5 :istore_3
6 :iload_3
7 : ireturn
```

```
OxDIFF<class_compute {
Method {
Name : compute_sum
Instr :
Del % 4
Add % isub 4
}end_meth
```

```
int compute(int,int) ;
Code
0 :iconst_0
1 :istore_3
2 :iload_1
3 :iload_2
//* Del  4
//* Add isub  4
4 :iadd
5 :istore_3
6 :iload_3
7 : ireturn
```

**Annotated Bytecode**

***Figure 4:*** *Bytecode annotation with update instructions*

## 6. ANNOTATION AND FUNCTIONAL REPRESENTATION OF BYTECODE

The DIFF file containing the update instructions is calculated at bytecode level and then sent to perform the update on-card. In order to ensure that we send the right one, we model its application on an initial version of bytecode P1 as annotations. The operation of annotating a bytecode with expressions indicating where an update instruction occurs and what is the operation involved can be defined recursively as an annotation function which transforms a program to an annotated program.

$Annot(\varepsilon, P) \equiv P$

$Annot([Upd_i|\Delta], P) \equiv let \ P' = Add_Annot_Line(Upd_i, P) \ in \ Annot(\Delta, P')$

The annotation of a program with an empty DIFF file ($\varepsilon$) is the program itself otherwise, the function iterates over the update operations ($Upd_i$) in the patch and adds a corresponding annotated line ($Add_Annot_Line(Upd_i, P)$) to the program. Figure 4 shows an annotated program obtained by the application of a DIFF file on an initial byte code. The annotations are represented as special commentaries. For example, *Del 4* : deletes the instruction at program counter $(pc)$ 4 and *add isub 4*, adds the instruction *isub* at $pc$ 4.

In our framework, we use a functional representation for both bytecode programs and annotation function. Figure 5 shows a fragment of the formalisation written in OCaml. We start by defining the data manipulated by the program (integers, objects and variables, then, we formalise the instructions of the sub language. The definition of an instruction is given by the name of a construct (representing the name of the instruction) followed by its arguments. For example, for the instruction *new*, we have the construct *New* taking an *Object* as argument and the instruction *putfield* is represented by the construct *Putfield* followed by a triple representing

```
♯ type object = Object;;
♯ type variable = Var;;
♯ type integer = Int;;
♯ type types = V of variable | O of object | I of integer;;
type jc_instr =
pop
| Store of variable
| Load of variable
| Const of integer
| Add
| Neg
| Return
| New of object
| If of integer
| Goto of integer
| Invoke of object * string * string
| Putfield of object * string * string
| Getfield of object * string * string;;
type bc_line = Line of int *jc_instr;;
type bc = Bc of bc_line list;;
type annot_line= AnnotL of bc_line * string;;
type annot_bc = AnnotBC of annot_line list;;
...
```

***Figure 5:*** *An extract of functional modelisation of bytecode*

the arguments: the class (*Object*) and the names of the type of the field and its name as strings.

A bytecode line is defined as a number (representing the program counter) with an instruction. The bytecode is represented as a list of bytecode lines. An annotated line is represented by the product of a bytecode line and a string representing the annotation. An annotated bytecode is a list of annotated bytecode lines. The result of this modelisation is used to derive specifications of updated programs.

## 7. VERIFICATION

Our approach for verification is based on the fact that the transformation of a bytecode (of its semantics) implies the transformation of its specification. In Hoare Logic (Hoare (1969)), a program $P1$ and its specification is represented by a triple $\{pre1\}P1\{post1\}$ where $pre1$ ($post1$) is the precondition (postcondition) of the program $P1$. A new version of this triple written off-card by the programmer is $\{pre2\}P2\{post2\}$ (a target triple). The DIFF file is performed with P1 and P2 and then sent to the card to perform update operations, meaning, obtaining a new bytecode and a new spacification. Our goal is to establish that the target triple and the obtained triple match.

### 7.1. Interpretation of the update

In order to formally define our update interpreter, we need to define some notions. In this interpretation, a state is modeled by a 3-tuple: $< Heap, Frame, Stack - Frame >$ which represents the machine state where Heap represents the contents of the heap, Frame represents the execution state of the current Method and, StackFrame is a list of frames corresponding to the call stack. A frame contains the following elements : the stack of operands $OperandStack$ and the values of the local variables $LocalVar$ at the program point $PC$ of the method $Method$ ( $< H, Method, PC, OperandStack, LocalVar >$ ). The definition of the update interpretation is based on the notion of step.

**Definition 1.** ***Step*** The semantics of an instruction (update instruction) is specified as a function step: $Bytecode_Prog * State * Specification- > State * StepName * Specification$ that, given a bytecode P, a state S and a specification SP, computes the next state S', the name of the next step and a new specification.

**Definition 2.** ***Java bytecode update interpreter*** We define now an update interpreter ($Upd_int$) which iterates over steps, take as parameters an annotated program in its functional representation, an initial state and an initial specification and relies on predicate calculus and update interpretation function to produce a new state and a new specification. The interpreter is defined as $Upd_int(BC, S) = (S', Sp')$ with $S = initial(BC, Sp)$ the function for defining an initial state for the execution of the bytecode $BC$ with the initial specification $Sp$. The Code $BC$ is given with its parameters and an initial heap. The result of the interpreter is a state $S'$ and a new specification $Sp'$.

**Definition 3.** ***Verified updated bytecode***

- Let $P1\ and\ P2$ be the first and the new version of a program and $P$ a patch,
- let $P2' = annot(P1, P)$ be the program obtained by annotation of P1 with P,
- let $f(P2')$ the functional representations of $P2'$,
- let $spec(P1) = (pre1, post1)$ the specification of $P1$ and $spec(P2) = (pre2, post2)$ the specification of P2,

We say that $P2'$ is a successfully verified update of $P1$ if and only if: $verification(spec(P2), spec(P2'))$ succeeds where $spec(P2')$ is obtained by predicate transformation on $f(P2')$ starting from $post2$.

### 7.2. Weakest precondition calculus

In this section, we define a bytecode update logic in terms of a weakest precondition calculus. The proposed weakest precondition (WP) considers that each (update) instruction has a precondition. An instruction with its precondition is called an

*Table 4: Defining rules for weakest precondition calculus for update operations*

| |
|---|
| **wp( Add_instr(pop,i))** $= (shift_exp^2(@E_i))$ |
| **wp( Add_instr(store x,i))** $= shift_exp^2(@E_i)(S(0)/x)$ |
| **wp(Add_instr(if L,i))** $= ((S(0) = 0) \Rightarrow shift_exp^2(E_L)) \wedge ((S(0) \neq 0) \Rightarrow shift_exp^2(@E_i))$ |
| **wp(Add_instr(load x,i))** $= unshift_exp(shift_exp(@E_i))(x/S(0))$ |
| **wp(Add_instr(const a,i))** $= unshift_exp(shift_exp(@E_i))(a/S(0))$ |
| **wp(Add_instr (new A,i))** $= unshift_exp(shift_exp(@E_i[create(H,A)/S(0), A :: H/H])$ |
| **wp(Add_instr(add,i)** $= (shift_exp^2(@E_i))[(s(1) + S(0))/S(1)]$ |
| **wp(Add_instr(neg,i)** $= (unshift_exp(@E_i))[-S(0)/S(0)]$ |
| **wp(Add_instr (getfield a f t,i) )** $= shift_exp(@E_i[(val(S(0),(a,f)))/S(0)]) \wedge S(0) \neq null$ |
| **wp(Add_instr(putfield a f t,i))** $= (shift_exp^3(@E_i))[H((S(0),(a,f)) := S(1))/H] \wedge S(0) \neq null$ |
| **wp(goto l1)** $= shift_exp(E_{l1})$ |

instruction specification and is noted as: $E_i : I_i$ where $I_i$ is the instruction and the expression $E_i$ its specification. This notation expresses that the precondition $E_i$ holds when the program pointer is at the program counter *i*. Table 4 shows the calculus of the WP rules for the update operations (inserting instructions).

**Functions and notations used.** The functions $shift_exp$ and $unshift_exp$ are used to express: the effect of pushing (popping) elements to (from) the stack $S$ and the effect of shifting an expression regarding to the stack elements due to the insertion of instructions. They are defined as follows:

$$shift_exp(Exp) = Exp[s(i+1)/s(i)\ forall\ i \in N]$$
$$unshift_exp = shift_exp^{-1}$$

The elements of the stack are represented by positive integers, the top stack is 0. The symbol $@$ is used to express the old specification associated to a position $i$: when we add an instruction at position $i$, the program and the specification are shifted from $i$ and then a new instruction is inserted. Its precondition is calculated with the specification of the instruction that was at position $i$ before the update.

In the rules, for the instructions $store\ x$, $load\ x$, and $pop$, a precondition is obtained, as in Hoare's assignment (Hoare (1969)) by substituting the right-hand side by the left-hand side in the postcondition. The precondition of an instruction $store\ x$ under a postcondition $E_{i+1}$ (the precondition of the following instruction) is given by: $shift_exp(E_{i+1})(S(0)/x)$ meaning that if the expression $E$ holds after the execution of $store\ x$ then it also holds for the top of the stack before storing it in $x$. The function $shift_exp$ is used to express that before the execution of the instruction, the top of the stack corresponding to the instruction at $i+1$ was at index 1.

Inserting an instruction, *e.g.* $store\ x$ at line $i$ means that the precondition of the old instruction at $i$ becomes the postcondition of the inserted instruction and thus the calculated precondition starts from this old postcondition ($@E_i$). The function $shift_exp$ is used twice ($shift_exp^2$) to express also the impact due to the insertion of the instruction on the specifications of the following instructions.

The instructions $new$, $putfield$ and $getfield$ are heap manipulating instructions. The function $create$ used in the instruction $new\ A$ returns a new object of type $A$ in the heap $H$. This obtained heap $(A :: H)$ replaces the old heap. The function $val$ used in the definition of getfield to get the value of the field $f$ of the class $a$ from the address (top of the stack). This value is then pushed on the stack. In $putfield$, the value of the field designated by the top of the stack is updated with the value at the second elements of the stack. The insertion of this instruction which pops two values implies three applications of $shift_exp$.

In order to establish semantical equivalence of a code written by the programmer and a program obtained by applying a DIFF file, we check the equivalence of the weakest precondition of an annotated program obtained by WP calculus and a precondition written by the programmer before DIFF file is performed.

### 7.3. Example

In order to illustrate how the logic works, we take the example of the function $abs$ that returns the absolute value of an integer taken as argument. This function is then transformed in order to get the double of the result in the initial calculus: for an integer given as argument, the new function returns the abstract value multiplied by two (modified abs). The specifications of the two functions are respectively:

$\{p = P\}\ abs\ \{(P \geq 0 \rightarrow result = P) \wedge (P < 0 \rightarrow result = -P)\}$

$\{p = P\}\ modified\ abs\ \{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\}$

In the specification, $P$ denotes the logical value at the entry and $result$ is the result of the function. Figure 6 shows the bytecode of the first version (a) and the second version (b) of the described function. The part (c) of the figure shows the DIFF file generated from the two versions. The last part of the figure (d) shows the bytecode of the function $abs$ annotated with update instructions. We notice that in this bytecode local variables are represented by integers: in $load\ 1$ for example, the number 1 means the local variable 1. The same notation is applied to other local variables.

In figure 7, The WP calculus is performed on the bytecode (without annotation) starting from the postcondition of the new version. The WP calculus is applied on the annotated bytecode as shown on figure 8. The specification for the update instructions are in bold. This example shows that we obtain the same precondition $\{P = v0\}$ which means that at the beginning of the calculus the logical value P is in the first local variable of the function. This result expresses the equivalence of the two bytecodes according to our definition of verified updated program.

## 8. RELATED WORK

Several studies have been conducted in order to use formal semantics to prevent type errors in bytecode. Our work extends the formalism presented in (Freund and Mitchell (1999)). This work defined semantics and a type system to study object initialization in bytecode. The original idea was developed in (Stata and Abadi (1999)) to study bytecode subroutines. In (Freund and Mitchell (2003)), the authors extended the work (Freund and Mitchell (1999)) to bytecode subroutines, virtual method invocation and exceptions. On another side, using predicate transformation to reason about bytecode properties has been studied in (Grégoire,Sacchini and Sivan (2008)). The authors presented a verification condition generator for bytecode formalized in the Coq proof assistant and based on weakest precondition calculus. Another work using weakest precondition to generate verification conditions from an annotated bytecode is presented in (Burdy and Pavlova (2006), Burdy et al. (2007)).

Our work is close to ( Freund and Mitchell (1999)) in the sense of the use of static semantics to analyze bytecode. The specificity of our work is the definition of semantics for updates. We use predicate transformation to reason about bytecode properties using existing tools for specification and proofs. Our bytecode logic for weakest precondtion calculus is inspired by (Bannwart and Müller (2005)). The authors present a Hoare-style logic combined with instruction specification in term of precondition for sequential bytecode. We adopted such instruction specification in our logic for weakest precondition for update operation.

In some studies, manipulating and analysing bytecode requires its modelisation in flexible representations suitable to the manipulation required. In (Puder and Lee (2009)), bytecode is represented by XML trees in order to use the technologies supporting XML to ease the injection and extraction of bytecode. In (Albert and al. (2007)), bytecode is represented by clauses written in Prolog to perform verification of bytecode programs. Generally, functional modelisation is used when the goal is to consider programs as mathematical models whose meaning is independent of runtime states. Therefore, it is possible to apply equational rewriting and reasoning to them (Guodong (2010)) and use several proof systems that are built on or uses functional languages in specifications.

## 9. CONCLUSION

In this paper, we proposed an approach for formalisation and verification of java bytecode updated programs. Our approach relies on four main concepts. We showed first how to use existing specification languages for Java and Java bytecode programs to write specification and transform them. Then, we defined a formal semantics which constitute a formal mean to establish the validity of update operations with regard to Java type safety. We proposed a functional representation of bytecode in order to model the application of update operations with the use of the notion of bytecode annotation. We presented a predicate transformation calculus based on weakest precondition for update operations to derive a specification for the annotated bytecode and showed how to establish the correctness of the update.

The approach presented is implemented using the OCaml language. Our study started with considering the system EmbedDSU but this is not restrictive, the framework proposed can be generalised to specification and verification of updated programs written in languages that are complied to bytecode. The use of the functional language and representation eases its integration with existing formal methods. Our immediate future work is to define WP calculus for instruction suppression. We plan to define another predicate transformation calculus (strongest postcondition) for update operation and the integration of our approach in an existing formal method supporting verification condition generation for functional programs.

```
int abs(int);
0: load 0
1: if 5
2: load 0
3: store 1
4: goto 8
5: load 0
6: neg
7: store 1
8: load 1
9: return
```

**(a)**

```
int abs(int);
0: load 0
1: if 7
2: const 2
3: load 0
4: mul
5: store 1
6: goto 12
7: const 2
8: load 0
9: neg
10: mul
11: store 1
12: load 1
13: return
```

**(b)**

```
OxDIFF<class_compute>
Method {
Name : abs
Instr :
Add const 2 2
Add mult 4
Add const 2 5
Add mult 7
} end_meth
```

**(c)**

```
int abs(int);
0: load 0
1: if 5
// Add const 2 2
2: load 0
// Add mul 4
3: store 1
4: goto 8
// Add const 2 5
5: load 0
6: neg
// Add mul 7
7: store 1
8: load 1
9: return
```

**(d)**

***Figure 6:*** *An example of an annotated bytecode (abs)*

int abs(int);
**0: load 0** $\{(P = v0\}$
**1: if 7** $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
**2: const 2** $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
**3: load 0** $\{(P \geq 0 \rightarrow s(0) * v0 = 2 * P) \wedge (P < 0 \rightarrow s(0) * v0 = -2 * P)\ \}$
**4: mul** $\{(P \geq 0 \rightarrow s(1) * s(0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * s(0) = -2 * P)\ \}$
**5: store 1** $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
**6: goto 12** $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
**7: const 2** $\{(P \geq 0 \rightarrow 2 * (-v0) = 2 * P) \wedge (P < 0 \rightarrow 2 * (-v0) = -2 * P)\ \}$
**8: load 0** $\{(P \geq 0 \rightarrow s(0) * (-v0) = 2 * P) \wedge (P < 0 \rightarrow s(0) * (-v0) = -2 * P)\ \}$
**9: neg** $\{(P \geq 0 \rightarrow s(1) * (-(s0)) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-(s0)) = -2 * P)\ \}$
**10: mul** $\{(P \geq 0 \rightarrow s(1) * s(0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * s(0) = -2 * P)\ \}$
**11: store 1** $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
**12: load 1** $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
**13: return** $\{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\ \}$

***Figure 7:*** *WP calculus on the modified function*

int abs(int);
**0: load 0** $\{(P = v0\}$
**1: if 5** $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
// **Add const 2 2** $\{(\mathbf{P \geq 0 \rightarrow 2{*}v0 = 2{*}P}) \wedge (\mathbf{P < 0 \rightarrow 2{*}v0 = -2{*}P})\ \}$
**2: load 0** $\{(P \geq 0 \rightarrow s(1) * v0 = 2 * P) \wedge (P < 0 \rightarrow s(1) * v0 = -2 * P)\ \}$
// **Add mul 4** $\{(\mathbf{P \geq 0 \rightarrow s(2){*}s(1) = 2{*}P}) \wedge (\mathbf{P < 0 \rightarrow s(2){*}s(1) = -2{*}P})\ \}$
**3: store 1** $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
**4: goto 8** $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
// **Add const 2 5** $\{(\mathbf{P \geq 0 \rightarrow 2{*}(-v0) = 2{*}P}) \wedge (\mathbf{P < 0 \rightarrow 2{*}(-v0) = -2{*}P})\ \}$
**5: load 0** $\{(P \geq 0 \rightarrow s(1) * (-v0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-v0) = -2 * P)\ \}$
**6: neg** $\{(P \geq 0 \rightarrow s(1) * (-s(0)) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-s(0)) = -2 * P)\ \}$
// **Add mul 7** $\{(\mathbf{P \geq 0 \rightarrow s(2){*}s(1) = 2{*}P}) \wedge (\mathbf{P < 0 \rightarrow s(2){*}s(1) = -2{*}P})\ \}$
**7: store 1** $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
**8: load 1** $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
**9: return** $\{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\ \}$

***Figure 8:*** *WP calculus on an annotated bytecode*

**REFERENCES**


Freund, S. N and Mitchell, J. C, (1999) *A type system for object initialization in the Java bytecode language*. In ACM Trans. Program. Lang. Syst., vol 21, pp.1196–1250.

Grégoire, B, Sacchini, J. L and Sivan, R, (2008) *Combining a verification condition generator for a bytecode language with static analyses*. In Proceedings of the 3rd conference on Trustworthy global computing, Springer-Verlag, pp.23–40.

Binder, W and Hulaas, J, (2005) *Java Bytecode Transformations for Efficient, Portable CPU Accounting*. In Electron. Notes Theor. Comput. Sci., Elsevier Science Publishers B. V. vol 141, pp.53–73.

Noubissi,A.C, Iguchi-Cartigny, J and Lanet,J. L, (2011) *Hot updates for Java based smart cards*. In ICDE Workshops, pp.168-173.

Burdy, L and Pavlova,M, (2006) *Java bytecode specification and verification* In SAC 2006, pp.1835-1839.

Burdy,L, Huisman, M and Pavlova,M, (2007) *Preliminary Design of BML: A Behavioral Interface Specification Language for Java Bytecode* In FASE 2007, pp.215-229.

Freund, S. N and Mitchell, J. C,(2003) *A Type System for the Java Bytecode Language and Verifier*. In J. Autom. Reasoning, vol 30, pp.271-321.

Hoare,C. A. R, (1969) *An Axiomatic Basis for Computer Programming*. In Commun. ACM, vol 12, pp.576-580.

Stata, R and Abadi, M, (1999) *A Type System for Java Bytecode Subroutine* In ACM Trans. Program. Lang. Syst., vol21, pp.90-137.

Dahm,M, (1999) *Byte Code Engineering*. InJava-Informations-Tage, pp.267-277.

Sakamoto, T, Sekiguchi, T and Yonezawa, A, (2000) *Bytecode Transformation for Portable Thread Migration in Java*. In ASA/MA, 2000, pp.16-28.

Bachrach, J and Playford, K, (2001) *The Java Syntactic Extender*. In OOPSLA 2001, pp.31-42.

Noubissi,A. C, Iguchi-Cartigny, J and Lanet, J. L, (2010) *Incremental Dynamic Update for Java-Based Smart Cards*. In Fifth International Conference on Systems, pp.110-113.

Burdy,L, Cheon,Y, Cok, D. R, Ernst, M. D, Kiniry,J. R., Leavens,G. T, Leino, K. R. M, and Poll, E. *An Overview of JML Tools and Applications* . In Int. J. Softw. Tools Technol. Transf., vol 7, pp. 212–232.

Bannwart, F and Müller, P, (2005) *A Program Logic for Bytecode*. In Electron. Notes Theor. Comput. Sci.vol 141, Elsevier Science Publishers B. V., 2005, pp 255–273.

Common Criteria,http://www.commmoncriteria.org

McGachey, P, Hosking,A. L and Moss, J.E.B, (2009) *Pervasive Load-Time Transformation for Transparently Distributed Java*. In Electron. Notes Theor. Comput. Sci., vol 253, Elsevier Science Publishers B. V., pp.47–64.

Puder, P and Lee, J, (2009) *Towards an XML-based Bytecode Level Transformation Framework*. In Electron. Notes Theor. Comput. Sci.,vol 253, Elsevier Science Publishers B. V., pp.97–111.

Boshernitsan, M and Graham,S. L, (2004) *iXj: Interactive Source-to-source Transformations for Java*. In Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, pp.212–213.

Guodong, L, (2010) *Formal verification of programs and their transformations*. PhD thesis, University of Utah, USA.

Lounas,R, Mezghiche, M and Lanet,J. L, (2012) *Towards a General Framework for Formal Reasoning about Java Bytecode Transformation* In Proceedings Fourth International Symposium on Symbolic Computation in Software Science, pp.63–73.

Noubissi,A. C, (2011) *Mise á jour dynamique et sécurisée de composants systéme dans une carte á puce*. PhD thesis, University of Limoges, France, 2011.

Albert, E, Gomez-Zamalloa, M, Hubert, L and Puebla,G, (2007) *Verification of Java Bytecode Using Analysis and Transformation of Logic Programs* . In Practical Aspects of Declarative Languages,2007, Springer Berlin Heidelberg,pp.124-139.

Neamtiu,I, Hicks,M, Stoyle, G and Oriol, M, (2006) *Practical Dynamic Software Updating for C*.In ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 7283, 2006.

Gupta, D, Jalote P and Barua, G. *A formal framework for online software version change*. Software Engineering, IEEE Transactions on, 22 (2):120131, 1996.

Orso,A, Rao, A and Harrold,M. J. *A technique for dynamic updating of Java Software*. In ICSM, 2002.

Hlopko, M, Kurs,J, and Vrany, J. *Towards a Runtime Code Update in Java an exploration using STX:LIBJAVA*. In proceeding of Dateso 2013.